Generalizing State Machine Replication (Preliminary Version)

E. Gafni (UCLA) and R. Guerraoui (EPFL)

August 6, 2010

Abstract

We show that, with k-set consensus, any number of processes can emulate k state machines of which at least one progresses. This generalizes the celebrated universality of consensus which enables to build a state machine that always progresses. Besides some interesting extensions and even potential "practical" applications, theoretically, a fundamental ramification of our main result, derived by considering the state machines to be interacting read-write threads, is a flagpole to the thesis that distributed computing is all about wait-freedom. We indeed show that the set of tasks that are read-write solvable "k-concurrently" , i.e., when concurrency goes below k, is the same set of tasks that are read-write solvable with k-set consensus.

Keywords: State machine replication, k-set agreement, universality, k-concurrency.

1 Introduction

The *state machine replication* approach is a general technique to make shared services highly available [12]. A service is modeled as a deterministic state machine. Processes hold each a copy of this state machine, to which they issue commands. To provide the illusion of sharing a single state machine, the processes use a *consensus* abstraction [6]. Consensus enables the processes to propose each a value and ensures that all agree on the same value. In the context of state machine replication, each instance of consensus is used to decide which command to execute next and hence make sure all commands are executed on the state machine in the same order: the very fact that the state machine is deterministic means that all its copies keep the same state. Consensus is said to be *universal* [10] in the sense its availability implies the availability of any shared service.

Yet, despite its universality, consensus is just the specific case of a more general abstraction: *k-set consensus* [5], where processes need decide on at most k different values. It is natural to ask what form of state machine replication we get if we generalize consensus to k-set consensus. Or, in other words, if a system does not dispose of consensus but only k-set consensus ($k > 1$), what form of state machine replication do we obtain? It is actually surprising that the distributed computing community has not yet tackled this question.

We show in this paper that k-set consensus is, in a precise sense, *k-universal*. Namely, we show that, with k-set-agreement, we can implement k state machines with the guarantee that at least one makes progress. We prove our result assuming a general shared memory model where processes can fail by crashing. In fact, we show an even more general result: processes can implement any number $m \geq k$ of state machines of which at least $m - k + 1$ progress. In fact, our generalization does not require to know k a priori: If we take k state machines and achieve j-set consensus, $j < k$, $k - j + 1$ machines will progress. When $m = k = n$ (the total number of processes), we obtain a new execution scheme of a read-write wait-free protocol - an "iterated" scheme [7] that provides a tie between iterated models and non-iterated ones. To get an idea of the technicality behind our generalized state machine replication protocol, recall that the basic idea underlying the classical construction is that each consensus instance is used to decide on the next command to execute. If k-set consensus is available instead of consensus, $k > 1$, the processes might get k different ordering of the commands. The challenges here are (a) to ensure that each ordering corresponds to exactly one state machine, (b) processes do not drop commands when switching from one machine to the other, and (c) at least one machine progresses at all processes.

Potential "practical" applications might be foreseen. Multiple state machines, implementing different services, one of which is guaranteed to progress, is better than one state machine that does not progress, say if consensus cannot be reached but k-set consensus can. In fact, multiple state machines, even implementing the same service, may provide for an interesting alternative behavior to a classical state machine replication scheme at the time when the system is not stable. Instead of blocking like a single machine will do, in our case at least one machine will progress. There is of course the danger that the state machines diverge from each other but many applications can tolerate divergence of view for a while. When the system stabilizes, these divergent views may be reconciled to continue with what is effectively a single view of the system. But there is actually much more: our result promotes the thesis that distributed computing is all about wait-freedom. A fundamental ramification of our generalization is indeed that the set of tasks that are read-write solvable "k-concurrently" , i.e., when concurrency goes below k, is the same set of tasks that are read-write solvable wait-free with k-set consensus, Hence, results about specific models of restricted concurrency can be deduced from (typically known) results about wait-free environments.

We first recall below the basic idea underlying state machine replication, then we present our general result, followed by its ramifications, and finally conclusions.

2 State machine replication: the classics

We assume here a standard read-write memory. Processes can be correct, in which case they execute an infinite number of steps of the protocol assigned to them, or they crash and stop any activity. The state machine replication approach works basically as follows (Figure 1). Every process has a copy of the state machine, denoted *sm* in Figure 1, as well as an ordered list of commands, denoted *commands* in that figure. A process picks one command at a time from this list; we also say that the process *issues* the command. The processes execute all commands on all copies of the state machines, in the same order, while preserving the original local order of the commands at each process. They do so by going round-based and, in each round, executing the command stipulated by the consensus object corresponding to that round. These consensuses form a list, denoted by *ConsList*, and exactly one object of the list is used in each round of the protocol. Basically, every process p first picks the next command it wants executed and proposes it to the next consensus instance. This, in turn, returns a command, not necessarily that proposed by p, but one proposed by at least one process. The command returned to a process p is then executed by p on its state machine: we simply say that p executes the command. To ensure that every process executes the commands in their original order, no process picks its next command unless it has not performed its previous one.

```
local data structures:
1 sm                                                        (* a copy of the state machine *)
2 comList                                                   (* a list of commands *)
3 passed = true                                             (* determines if the process executed its previous command *)
shared data structure:
4 ConsList                                                  (* a list of shared consensus objects *)
forever do:
5 if passed then com1 = comList.next()                      (* pick the next command *)
6 cons = ConsList.next()                                    (* pick the next consensus object *)
7 com2 = cons.propose(com1)                                 (* agree on the next command *)
8 sm.perform(com2)                                          (* execute the agreed upon command *)
9 if com2 = com1 then passed = true else passed = false     (* test if own command passed *)
```

Figure 1: State machine replication

The correctness of the protocol of Figure 1 lies on three facts. (1) If a process p executes command c, then c was issued by some process, and if p has issued c, then p has executed every command issued by p and preceding c. This follows from the fact that a process does not issue a new command unless it has executed its previous one. (2) If a process p executes command c without having executed command c', then no process q executed c' without c. This follows from the fact that the processes execute the commands output by the consensuses and these are invoked by the processes in the same order. (3) Every correct process executes an infinite number of commands on the state machine. This follows from the liveness of consensus: every invocation to consensus by a correct process returns a command to that object.[1]

[1]Our simple protocol does not guarantee fairness. Consensuses could always return the commands proposed by the same process. To ensure fairness, processes need to help each other: namely, when a process issues a command, it writes it in shared memory; processes would now propose sets of commands (their own and those of others) to consensuses; accordingly, a consensus would return a set; the set would be the same at all processes which would execute the commands in the same deterministic order. For presentation simplicity, we omit helping and fairness.

3 Generalized state machine replication

What if, in classical state machine replication (Figure 1), some consensuses are *faulty* and, instead of returning a single value to all, return k different values? This question is typically viewed as an issue of safely implementing consensus rather then an issue of the state machine replication protocol. The key idea behind our generalization is to view this as an issue of the replication protocol.

In the following, we generalize state machine replication to show that, with k-set consensus, i.e., if k distinct values can be returned by the agreement abstraction, the n processes can execute k state machines one of which at least progresses.

3.1 Vectors of commands and consensuses

In our generalized setting, the processes have at their disposal a k-vector of commands that they are supposed to execute on their local copies of the k state machines: entry j in the vector to be executed on machine sm[j]. We assume k-set consensus in its vector form [1]: basically a vector of consensuses, denoted *ConsVector(1,..,k)*, that takes as input commands in the form of k-vectors, and returns, to each process, a k-vector composed of *nil* values and commands among those proposed (we assume here that commands cannot be *nil*). Any two commands returned at the same position (i.e., by the same consensus object) to any two processes are the same. Yet, one process might get a command returned at position i and *nil* at position j, whereas another process might get some command at position j and *nil* at position i.

To get a sense of the technical difficulty behind our generalization, consider first a naive protocol resulting from replacing, in Figure 1, consensus with k-vector consensus, and assume that, in round r, a process p executes on state machine sm[i] the command obtained at position i from the k-vector consensus. If p obtains *nil* at position i in r, then p does not execute anything on state machine sm[i] in round r. Clearly, such a protocol would guarantee that at least one state machine will progress since the consensus vector will return at least one non-nil value and at least one command will be executed in every round. Yet, consider the following scenario. Assume p gets a command c at position 1 in round 1 (after proposing its command vector): p will then accordingly execute c on sm[1]. In the meantime, assume process q has obtained a command c' at position 2 and executed c' on sm[2]. Assume now that, in round 2, p obtains a command at position 2: according to our (naive) protocol, process p executes it on its state machine sm[2]: yet this will violate safety for p ignores that q already executed c' on sm[2] in round 1.

Intuitively, the issue should be sorted out by having every process announces what command it has executed before proceeding to the next round: say q would need to inform p that it has executed c' on sm[2] in round 1. This is not entirely trivial and needs to be synchronized with the action where p needs itself to execute a command on sm[1]. (This is reminiscent of our question above about *faulty* consensuses.)

3.2 Filtering with adopt-commit

To sort out the synchronization issue among state machines, *adopt-commit* objects [7] come in handy. The specification of such an object is as follows. Every process proposes a command, and obtains a command, either in a *committed* or *adopted* status. If an adopt-commit object returns a committed command to a process, it returns the same command (committed or adopted) to every other process. Furthermore, if all proposals to the same adopt-commit object are the same command, then this command is committed. Such objects can be implemented with a standard read-write memory.

We use a vector of adopt-commit objects at each round, and this acts as a synchronization *filter* through which processes go before executing commands on their state machines. Each process, after obtaining an output from the consensus vector, goes through adopt-commits to validate what needs to be executed on its state machines: in short, a process only executes commands that are committted and keep those adopted for the next round. The order according to which adopt-commits are accessed and how adopted command are handled are crucial aspects of our protocol.

1. *Exploit successes first.* To ensure liveness, a process p, at round r, accesses first the adopt-commit objects corresponding to the positions of the commands returned by the consensus vector at r. Process p can access them in any order or simultaneously. Subsequently, p proceeds to the rest of the positions at which is was returned *nil* and proposes the original entry values to the consensus vector.

   This ensures that at least one process will commit a command in every round. Indeed, for an adopt-commit object not to commit, it has to be concurrently invoked with at east two distinct values. The first process p to return from any of the adopt-commit, by virtue of being first, must commit the command. No processes q can prevent it by inserting a distinct command concurrent with p, as then, q's command was not returned there, and q already finished the commit-adopts of its returned command, contradicting the fact that p was the first to return from any adopt-commit..

2. *Remember commitments.* To ensure safety, a process p might need to execute two commands on the same machine in the same round. A process p might indeed adopt a command c in round r for position i, then commit another command c' in round $r+1$ for that same position i. This might happen if another process q committed c at r and then moved to propose and commit c' at $r+1$. Should p execute c' without having executed c, p would violate safety.

   In our scheme, when q commits a command c in round r, then moves to round $r+1$, q encodes in c' the fact that c was committed before c': hence, in round $r+1$, p will decode that information from c', then execute c before c'. In fact, p executes c even if it only adopts c' in round $r+1$. Recall the c cannot "get lost" as any process that did not commit c in round r must have adopted c at round r. Hence, all proposed values to the commit-adopt at position i at round $r+1$, which are not c, are values which encode the very fact c has been committed at round r.

Our generalized state machine replication protocol is depicted in Figure 2 where we denote the vector of adopt-commit objects by *ACVector(1,..k)*. Also, we denote the k-vector of commands available to a process by *comVectList*: For presentation simplicity, we assume the following:

- We can test whether the returned value is committed (resp. adopted) simply using a function *committed(c)* (resp. *adopted(c)*).
- We assume that a process can pick the next command using function *next()* but also recall the last command picked using function *current()*.
- We assume that a process can encode in a command c' the fact that the process has committed c, simply by writing $c'.add(c)$, and a process can check that by simply testing if $c < c'$.

```
local data structures:
1  smVect[]                                              (* a vector of state machines *)
2  comVectList                                           (* a list of command vectors *)
3  for j = 1 to k do: comVect[j] = comVectList[j].next()  (* pick the first command vector *)
shared data structures:
4  ConsVectList                                          (* a list of vector consensus objects *)
5  AConsVectList                                         (* a list of vector adopt-commit objects *)
forever do:
6  consVect = ConsVectList.next()                        (* pick the next consensus vector *)
7  comVect1 = consVect.propose(comVect);                 (* select a new vector *)
8  aconsVect = AConsVectList.next()                      (* pick the next adopt-commit vector *)
9  for i = 1 to k do:
10   if comVect1[i] ≠ nil then:
11     comVect2[i] = aconsVect[i].propose(comVect1[i])   (* try to commit commands *)
12 for i = 1 to k do:
13   if comVect1[i] = nil then:
14     comVect2[i] = aconsVect[i].propose(comVect[i])    (* try to commit commands *)
15 for i = 1 to k do:
16   if comVect2[i] > comVect[i] then sm[i].execute(comVect[i])   (* catch-up *)
17   if adopted(comVect2[i]) then comVect[i] = comVect2[i]        (* keep the command for next round *)
18   else
19     sm[i].execute(comVect2[i])
20     if comVect2[i] = comVectList[i].current() then comVect[i] = comVectList[i].next()
21     else comVect[i] = comVectList[i].current()
22     comVect[i].add(comVect2[i])                       (* remember the committed value *)
```

Figure 2: Generalized state machine replication

3.3 Correctness

Theorem 1. *If a process p executes command c on its state machine sm[i], then c was issued by some process, and if p has issued c, then p has executed every command c' issued by p before c.*

Proof. (Sketch) A process executes a command c only if c has been returned to the process by an adopt-commit object. By the protocol of Figure 2, c must have been proposed to some adopt-commit object, and hence returned by a vector consensus. In turn, this can only return commands that have been proposed, and hence issued by a process. Furthermore, no process issues a new command c unless the process has executed all commands preceding c. □

Theorem 2. *If a process p performs, on its state machine sm[i], command c without having performed command c', then no process q performs c' without c on its state machine sm[i].*

Proof. (Sketch) Assume a process p executes, on its state machine sm[i], command c at some round r. By the protocol of Figure 2, this means adopt-commit object aconsVect[i] has committed c at round r or $r-1$. Assume furthermore that process p did not execute c' before c. This means that not adopt-commit object aconsVect[i] has committed c' at round $r' < r-1$ Hence, no process q can execute c' without having executed c on state machine sm[i]. □

Lemma 1. *If a process p commits command c in round r for state machine sm[i], then every process which finishes round $r+1$ executes c on state machine sm[i].*

Proof. (Sketch) Assume process p commits command c in round r for state machine sm[i], i.e., adopt-commit object aconsVect[i] returns command c in a committed status in that round. By the specification of adopt-commit, aconsVect[i] returns command c (either in a committed or aborted status) to all processes that invoked it in round r. Hence, all processes which start round $r+1$ either (a) executed c on sm[i] in round r and start round $r+1$ with a command c' such that c precedes c, or (b) start round $r+1$ with proposal c itself. In any case, any process that did not execute c in round r will, in round $r+1$, knows about c having been committed and execute it. □

Theorem 3. *An infinite number of commands are executed on at least one state machine at all correct processes.*

Proof. (Sketch) Assume at least one process is correct. Assume by contradiction that there is a round at which no process executes a command on a state machine. This means that no adopt-commit object returns a committed command. Given that the protocol of Figure 2 has no wait statement, every adopt-commit object must have had two different concurrently proposed values. This means that all processes obtained different values from the consensus vector. Consequently, all processes started at different adopt-commit objects. This is in contradiction with the fact that every adopt-commit object has two different concurrent proposals. Hence, at least one process commits a command on at least one machine in every round. This follows from the order according to which processes access adopt-commit objects. By Lemma 1, all correct processes execute a command on at least one machine in every two rounds. Hence, there is at least one state machine on which all correct processes execute an infinite sequence of commands. □

3.4 Extension to any number of machines

Our generalized state machine replication protocol uses k-set consensus to reduce the a priori *disagreement* among the processes in every round, i.e., restrict the set of different proposed command vectors in each round. With m processes, the worst disagreement we can get is m. In fact, the a priori disagreement among the processes can be viewed as a dynamic notion, that eventually stabilizes to some value j. We will later show how to make good usage of that notion.

Interestingly, the protocol of Figure 2 can be used, almost as such, to build m machines such that, when the disagreement among the processes stabilizes to be bounded by j, at least $m-j+1$ machines progress. The only abstraction that needs to be extended is the vector consensus. In its original form [1], with a disagreement of j and m, $m > j$, independent consensuses, only one command is guaranteed to be returned. What we need is to guarantee $m-j+1$ non-*nil* returns.

We explain now how to obtain this extension of [1] which, we believe, is interesting in its own right. Assume at most j different vectors are proposed, i.e., disagreement is j. (Two vectors are distinct if they differ on at least one of their entries and a consistent ranking among proposed m-vectors is assumed.) Every process p proceeds through m phases. In phase 1, p posts its m-vector input to shared memory. In phase i, p posts the m-vector proposal it obtained at phase $i-1$, as we explain now. Process p then looks how many distinct m-vectors were posted at phase i. If the number is greater than 1, p proceeds to phase $i+1$ with the lowest rank vector it observed at phase i. Else, if p sees only a single m-vector, the one it posted, p posts the pair (i, v_i) in shared memory, where v_i is the value of the ith entry in its m-vector. Process p then proceeds with this vector to phase $i+1$. After finishing phase m, p considers all the pairs $(index, value)$ it observed and, for every $index$, it returns the $value$ in $(index, value)$. Clearly, the condition to post (i, v_i) is

to observe a single m-vector in phase i, and thus only one vector may be observed if at all. Now, if no value is posted for phase i, then the highest ranked vector posted in phase i will not appear in any subsequent phases, and thus at most $k-1$ phases might have no posted value by the time any process terminated phase m.[2] Using our extended vector consensus abstraction, we obtain the following extension of Theorem 3:

Theorem 4. *Assuming any number of processes of which disagreement stabilizes to j, the protocol of Figure 2 emulates any number of state machines m with a guaranteed progress of at least $max\{m-j+1, 0\}$ machines.*

4 Ramifications

4.1 Read-write threads as state machines

Commands applied to a state machine can be thought of as operations applied to some shared object. Multiple state machines bring a new dimension: we can think of the machines interacting with each other, namely *threads* of alternating *write*s and *read*s to shared memory. Since each *write* following a *read* is the function of what the *read* returned, and all processes hold copies of the state machines locally, then a thread can be viewed as an initial *write*, followed by an infinite sequence of *read*s. The value of the first *write* of a thread, which will be its first linearized command, will be the input to the thread. From there on, the *read* commands return the state of all the machines. Thus in each round, the entry i of the m-vector proposal p submits is either an adopted value, if in the previous round p adopted a value for machine i, or the the state of the replicated machines as viewed locally by p, if a value was committed for machine i in the previous round. As local copies may differ, nevertheless, each is a valid *read* value, only taken at different time. Any value that succeeds to commit is a valid value for the simulated asynchronous threads executed by the state machines.

Let T be a task on a set of processes P, $|P| = n$, and let Π be a read-write wait-free protocol solving T. Protocol Π consists of threads $\Pi = \pi_1, ..., \pi_n$ where, π_i is to be executed by p_i, exclusively. Yet, a thread can also be viewed as a sequence of commands and we can think of π_i as a state machine to be executed by all, the commands happened to be *read*s and *write*s. This view encounters two difficulties.

1. How can we "start" π_i? i.e., how do we know the input of p_i to use as a value of its first *write*.
2. π_i is finite, state machine accepts unbounded number of commands, what do we do after it terminates?

We can dispose of the two difficulties by applying "null" commands to threads for which there is no input, and similarly apply "null" commands to π_i after it terminated.

In executing Π, we are interested in the advancement of threads that have not terminated and for which we have inputs. A state machine advancing by "null" commands is not progressing the execution. But a progress in the execution is a function of the participating processes. They are

[2] Interestingly, we could also build the same extended vector consensus abstraction using our own generalized state machine replication approach and [1] as black-box. The technique in [1] guarantees that, with j disagreement, the vector consensus will return a value at an index not above j. Thus we can implement with our state machines as explained in the next section dynamically j threads with at most $j-1$ faulty. These threads can then use *safe* agreement [4] to return $m-j+1$ values for m independent consensuses.

the ones that arrive and need an output. When each process either did not arrive or arrived and obtained an output, no progress is required. As long as p_i has not invoked T, all m-vectors propose "null" for machine i. When p_i arrives, it will participate trying to commit its input to π_i on machine i until p_i succeeds. Only then can the execution of π_i progress. Lest this input command will be forever pre-empted by "null", and p_i proposal vector interfere with the execution of the machines that can be moved by all, we amend the round structure. As long as p_i's input is not committed at π_i, at the beginning of each round it starts by posting its input, before it proceeds as usual. Every process at an end of any round looks for posted inputs, and put the input next on its list of command to π_i if it has not done it before. If p_i's proposal vector affects what the vector consensus returned then all will observe p_i's input and put it on their list. As the number of distinct proposal values is bounded by the number of executing processes, Theorem 4 guarantees that as long as a processes arrived and has not terminated, at least one of the threads of Π will advance by committing value commands rather than "null" commands.

We thus have obtained a new method of executing threads. Rather than by a standard execution model in which a process executes a unique thread of Π in exclusion, now the only association between p_i and π_i is that p_i supplies the input to π_i, and p_i departs once π_i has terminated. Below we will apply our state machine replication scheme of executing a protocol to obtain numerous applications.

4.2 Iterated vs non-iterated computations

We note first that, as our state machines execution proceeds in an iterated fashion [7], the above is an alternative argument that the standard wait-free shared memory model, where variables are not restricted to be used in a single round, and the iterated one, where variables are private to a round and in each round a process executes a number of steps which is independent of the protocol it executes, are equivalent in their power to solve tasks.

Furthermore, if the system is *t-resilient*, the processes wait at the beginning of each round until they see at least $n - t$ processes providing proposal vectors: it is also easy to see inductively that these can be made into at most $|P|-(n-t)+1$ distinct proposal vectors. Consequently, by Theorem 4 at least $n - t$ threads will advance producing a t-resilient execution of the threads. This extends the equivalence of the iterated model and the read-write shared memory model [3] to t- resiliency.

4.3 k-set consensus = k-concurrency

Let T be a task on n processes solvable by read-write and k-set consensus. We show that T is solvable without k-set consensus if at all times the number of pending task invocations is bounded by k. An invocation of T is pending for p_i if p_i provided its input but the task execution did not provide an output to p_i yet. The number of pending T invocations is the number of processes that have pending invocations of T. Let Π be a protocol to solve T within which processes can invoke any of many copies as required, of k-set consensus objects. An invocation to a consensus object COK_i is *pending* if the invocation occurred but a corresponding response with an output value did not arrive yet. As long as it does not arrive the invoking process is blocked from progressing. Consequently, if the number of pending invocations of T is at all times bounded by k, then we are guaranteed that at most k invocations to any k-set consensus object might be pending at any time. We call an execution of T with at most k concurrent pending task invocations a *k-concurrent* execution.

We produce the read-write code, with no access to k-set consensus, that k-concurrently will solve T. For this we replace each invocation of a k-set consensus object COK_i in Π with a read-

write code section. A MWMR variable ki in shared memory, initially $\perp$, is dedicated to replacing COK_i. A process that executes Π and gets to a command to invoke COK_i with value v, first reads ki. If it reads a value val that is not $\perp$ it returns val in the simulated invocation of COK_i. If it reads $\perp$ it writes v into ki and returns v.

The correctness of the modified code to solve T k-concurrently follows from the observation that at most k processes that invoke COK_i can observe ki as $\perp$. This follows from the fact that any process that observes ki as $\perp$ will not terminate before writing its value into ki. Since the concurrency is k at most k such pending writes can occur and consequently all processes that invoke COK_i will return one of these k values submitted by the processes that read ki as $\perp$.

The harder direction is to show that if Π is a read-write protocol to solve T k-concurrently then there exists a read-write code with k-set consensus to solve T wait-free. For this we use the state machines. We take the k-concurrent protocol Π. Using k-set consensus, the protocol Π will be executed k-concurrently. We consider a fixed set of k state machines as an intermediary between the processes and the thread of Π. Call these k intermediaries - servers. Instead of processes executing the threads of Π as state machines, they will execute the servers as state machines. The servers, in turn, will execute threads of Π as state machines in a special way. They will execute a BG agreement for each command in the threads of Π. The reason for the difference in the method of executing state machines, is that we expect the number of processes to be larger than the number of servers, but the number of faulty servers to be always smaller than the number of active threads of Π.

We make a slight change to the BG agreement protocol [4] to accommodate such a setting. Normally, once a process gets to the end of the agreement it forks [4]. Now on the other hand if it got to the "wait" statement and the agreement value is available, it does not fork but rather continues to the next command of the same thread. If it executed the last *read* of a thread, it goes to another thread in a round robin until it finds a thread on which the agreement has a value, and stick with the thread again. It will continue on the thread until termination or until it waits on an agreement protocol with no value, in which case it forks.

If the number of BG servers is k it is easy to see that at most k treads can be in the middle of execution at a time. The last BG server to terminate agreement will not fork and stick with the thread. Thus, a thread, once started will until termination have some server associated with it. By Pigeon-Hole at most k threads might have started but not terminated. But we do not have k servers, we have m servers. We notice that, with k-set consensus by Theorem 4 at least $m-k+1$ servers will progress. It is easy to see how to implement,by the standard method of [4], k BG processes with at most $k-1$ faulty, out of m BG processes with at most $k-1$ faulty, as we are guaranteed that at least $m-k+1$ of the m will progress.

When the number of processes j executing the servers is less than k, enough servers are non-faulty so that at least one of the j threads advances. This follows from Theorem 4 as, with $j < k$ processes, the bound of the disagreement is j rather than k. We do not have to worry now about simulating just $j < k$ machines as the number of a available threads to advance on is less than k, and progress will happen nevertheless as at most $j-1$ BG servers at most might be faulty.

As with t-resilience, another ramification of this subsection is that the iterated model with k-set consensus is equivalent to the standard one with k-set consensus.

4.4 k-set consensus = k-obstruction-free = k-1-active resiliency

A task T is solvable *k-obstruction-free* if progress is required only when contention is reduced to k or less [9], i.e., when at most k read-write threads try to progress. This is like *k-concurrency* only that program-counters of the read-write code Π to solve T obstruction-free may be all over

the place, at the time that the condition is invoked. This in contrast with k concurrency, where at least $n-k$ program-counters are at the beginning or end of a thread.

To see that T is solvable with k-set consensus, notice that a k-concurrent execution is by definition k-obstruction-free. On the other hand, with k-obstruction-freedom, k-set consensus is solvable.

The condition $k-1$-*active resiliency* requires progress only when at most $k-1$ of the processes that invoke the task and have not terminated may fail. If a task T is solvable with k-set consensus then T is solvable $k-1$-active resiliently since, with at most $k-1$ processes failing, k-set consensus is solvable. Obviously if a task is solvable with $k-1$-active resiliently, it is solvable with k-set consensus, as this can implement a k-concurrent execution, a special instance of $k-1$-active resiliency.

4.5 The multiplicative-power of (x, y)-set consensus

A (x, y)-set consensus object allows x processes to solve y-set consensus, $y < x$. Recently [11], it was observed that in the BG simulation, if n BG simulators are given $(x, 1)$ consensus, then essentially x BG simulators can be considered to be a single simulator. This relies on the fact that *test-and-set* is solvable when $x \geq 2$, which in turn allows for the acquisition of a "port" in the $(x, 1)$ object. What if we have $(x, 2)$ objects? Can we essentially equate x BG simulators with just 2simulators?

Interestingly, our generalized state machine replication scheme means that, with $(x, 2)$-set consensus, any n BG simulators can solve $2(n \ div \ x)$-set consensus. This can be accomplished by streaming processes through each of the $\binom{n}{x}$ objects, each hard wired to a fixed distinct set of x processes, in which each has a port. The BG simulators are now our n processes and they can drive $m = 2(n \ div \ y)$ "BG-servers" with at least one progress.

4.6 Renaming with k-set consensus

Proving that that any number of processes with k-set consensus available, can rename with $k-1$ overhead positions was so far challenging [8]. Now it is trivial to see that the overhead is $k-1$ because, at each point in time we have at most k processes going wait-free while the others announced their output.

5 Concluding remark

We just outlined few of the ramifications of our generalized state machine replication approach. While many papers devise specific constructions to get results when k-set consensus is available, our generalized state machine replication approach gives one tool to potentially get all. Rarely is one so lucky. Yet, this should not be surprising as this is obviously known to be the case with consensus. There seems indeed not to be any property of consensus which does not have a counterpart with k-set consensus.

References

[1] Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., Simultaneous Consensus Tasks: A Tighter Characterization of Set-Consensus. ICDCN 2006: pp. 331–34.

[2] Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R. Renaming in an Asynchronous Environment Journal of the ACM 37(3): 524-548, 1990.

[3] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-Free Computations. PODC 1997: pp. 189–198.

[4] Borowsky E., Gafni E., Lynch N. and Rajsbaum S. The BG Distributed Simulation Algorithm. Distributed Computing, 14(3):127–146, 2001.

[5] Chaudhuri S. More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. Information and Computation, 105:132-158, 1993.

[6] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. Journal of the ACM, 32(2): 374-382, 1985.

[7] Gafni E. Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. PODC 1998: 143-152.

[8] Gafni E., Renaming with k-Set-Consensus: An Optimal Algorithm into n + k - 1 Slots. OPODIS 2006: pp. 36-44.

[9] Herlihy, M., Luchangco, V., and Moir, M., Obstruction-Free Synchronization: Double-Ended Queues as an Example. ICDCS 2003.

[10] Herlihy M.P. Wait-Free Synchronization. ACM Transactions on programming Languages and Systems, 11(1):124-149, 1991.

[11] Imbs D. and Raynal M. The Multiplicative Power of Consensus Numbers. PODC 2010.

[12] Lamport L: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7): 558-565, 1987.